Attorney Dkt. No. 01 P 15956 US

Express Mail mailing label no: EL683752245US

APPLICATION FOR LETTERS PATENT OF THE UNITED STATES

NAME OF INVENTORS: Heonchul Park
10146 Alpine Drive, #1
Cupertino, CA 95014

TITLE OF INVENTION: THREE INPUT VARIABLE SUBFIELD COMPARATION FOR FAST MATCHING

TO WHOM IT MAY CONCERN, THE FOLLOWING IS A SPECIFICATION OF THE AFORESAID INVENTION

THREE INPUT VARIABLE SUBFIELD COMPARATION FOR FAST MATCHING

Heonchul Park

FIELD OF THE INVENTION

[0001] The present invention relates to memory management units, and more particularly to a method and apparatus for performing fast address comparison.

BACKGROUND OF THE INVENTION

[0002] Modern computer systems allow multiple processes (e.g. software programs) to run simultaneously on a central processing unit (CPU). These processes store and retrieve information from memory within the computer system. This memory is accessed by means of a physical address, i.e., an absolute coordinate system by which individual memory locations are identified. In the interests of efficient memory allocation and for the ease of software design, individual processes are not allocated the physical addresses of the physical memory. Rather, individual processes are allocated a virtual address representing the portion of the memory resources which are available to that process. The total allocated virtual addresses (the virtual address space) may represent a larger space than the actual physical memory, allowing individual processes to operate as if more memory than the actual physical memory is present in the computer system.

[0003] A memory management unit (MMU) mediates between the virtually addressed memory accesses requested by each process and the physical addresses of the available physical memory storing the desired data. An MMU is a device or circuit that supports virtual memory and paging by translating virtual addresses into physical addresses. Each process running in a system with an MMU receives a portion of a virtual address space. Typically, the virtual address space is divided into pages of 2^N bits each, each process receiving a number of pages. Each virtual address accessed by a process running on the CPU is split into a virtual page number (VPN), comprising the most significant bits of the virtual address, and an offset within that page, comprising the N least significant bits of the virtual address. Thus, if an address is 32 bits and a page size is 4KB ($2^N=2^{12}$), then the page offset is defined as the rightmost (least significant) 12 bits (N) of the address and the virtual page number is defined as the remaining (most significant) 20 bits in the 32 bit virtual address.

[0004] The MMU contains a page table that is indexed by the virtual page number. While the offset is left unchanged by the MMU, the virtual page number is mapped through the page table to a physical page number (PPN). Each page table entry (PTE) has a stored virtual page number and a corresponding physical page number. The page table provides a physical page number corresponding to a stored virtual page number matching an applied virtual page number. This physical page number is recombined with the page offset to create a physical address, which is an actual location in physical memory (e.g. RAM). For example, with an 8 bit address and a 4 bit (2^2) page size, the most significant 4 bits of a virtual address are the virtual page number and the least significant 4 bits are the page offset. A particular process generates a first address including a first virtual page number, which is applied to a page table of an MMU. Through the page table lookup, the MMU determines a first physical page number from the virtual page number for the particular process. The page offset (least significant 4 bits) is combined with this first physical page number to generate a physical address of a location in memory.

[0005] To save time in translating virtual page numbers to physical page numbers, the most recently used page translations may be cached in a translation look-aside buffer (TLB). Translation look-aside buffers typically have an associative memory, with comparators for each word, so that the translation can be retrieved in one clock cycle. If the virtual page number is not found in the translation look-aside buffer, then the MMU page table is checked for the virtual page number. A typical translation look-aside buffer has four cached page translations. For some applications, a virtual address is calculated from the sum of two operands. As a result, the translation of a virtual address to a physical address first requires calculation of the virtual page number from the most significant bits of the sum of the two operands, and then requires translation by a translation look-aside buffer or page table to a physical address.

[0006] Figure 1 is a block diagram of a conventional system 100 for the calculation of a physical address from two 32 bit operands. An adder 110 receives a first 32 bit operand OPA and a second 32 bit operand OPB, combining them to form a virtual page number VPN_IN and a page offset PO. A translation look-aside buffer 120 includes four rows and two columns, each row in the first column storing a virtual page number (e.g. VPNs 130, 131, 132, and 133) and the corresponding row in the second column storing a physical page number (e.g. PPNs 140, 141, 142, and 143). For example, the first row in translation look-aside buffer 120 contains a virtual page number 130 and a corresponding physical page number 140. Thus, translation look-aside buffer 120 maps an applied virtual page number matching virtual page number 130 to a physical page number physical page number 140. In this example, translation look-aside buffer 120 provides matching physical page number 140 as a physical page number PPN_OUT.

[0007] The page offset PO generated from the sum of operands OPA and OPB is concatenated with physical page number PPN_OUT to generate a physical address. Specifically, translation look-aside buffer 120 receives virtual page number VPN_IN and compares virtual page number VPN_IN to each of VPNs 130, 131, 132, and 133. If virtual page number VPN_IN matches one of VPNs 130-133, translation look-aside buffer 120 provides the corresponding one of PPNs 140, 141, 142, and 143 as PPN_OUT. A concatenator 115 combines page offset PO with physical page number PPN_OUT to generate the physical address PA.

[0008] The process of adding 32 bit numbers and then applying them to a translation look-aside buffer takes a certain amount of time. Because the addition of operands OPA and OPB (e.g. a 32 bit addition) within adder 110 occurs in a sequential fashion, the overall time taken to determine a physical address from a virtual address is delayed by the amount of time taken to complete the serial addition of the operands prior to their comparison to the cached VPNs. Additionally, adder 110, which can accommodate 32 bit operands OPA and OPB, requires a large amount of circuit area. Hence, it would be desirable to reduce the amount of time and space required to produce a physical address from a virtual address formed by two operands.

SUMMARY

[0009] Accordingly, a fast system and method for generating physical page numbers (PPNs) in accordance with one embodiment of the present invention includes exploiting a relationship between the sum of two operands and a matching virtual page number (VPN). Direct parallel application of sum and carry bits from a portion of each operand as well as a pre-stored virtual page number bit to a plurality of comparators minimizes the time required to process a sum of a large operand as well as, in some cases,

reducing the circuit area required for the operation. In one embodiment, two operands generating a virtual page number with a known size are applied to a bit adder along with a pre-stored virtual page number. The sum and the logical NOT of the carry generated by the bit adder are compared to determine if a match exists between the sum of the two operands and the pre-stored virtual page number. A match indicates that the applied virtual page number matches the virtual page number stored in the page table.

[0010] In another embodiment of the present invention, a portion of two variably-sized operands generating a virtual page number are applied to a bit adder along with a pre-stored virtual page number to generate a concatenated bitwise sum and a concatenated bitwise carry. Additional portions of the two variably-sized operands are augmented with additional bits, called indicator bits, and applied to a word adder to generate a wordwise sum and a wordwise carry bit. The concatenated bitwise sum, the logical NOT of the concatenated bitwise carry, and the inverse of the wordwise carry bit are applied to a comparator to determine if a match exists between the sum of the operands and the pre-stored virtual page number. As stated above, a indicates that the applied virtual page number matches the virtual page number stored in the page table.

[0011] The present invention will be more fully understood in view of the following description and drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0012] Fig. 1 is a simplified diagram of a conventional system for calculating a physical page number from two operands.

[0013] Figure 2A is a block diagram of a compare system for an MMU in accordance with one embodiment of the present invention.

[0014] Figure 2B is a block diagram of an AND tree comparator for an N bit address and 64K page size in accordance with one embodiment of the present invention.

[0015] Figure 2C is a block diagram for a one-bit adder in accordance with the embodiment of Figure 2A.

[0016] Figure 3 is a block diagram of a compare system in accordance with another embodiment of the present invention.

[0017] Figure 4A is a block diagram of a compare system in accordance with another embodiment of the present invention.

[0018] Figure 4B is an example of an augmentation and the wordwise addition of two operands OP1 and OP2 in accordance with an embodiment of the present invention.

[0019] Figure 5 is a schematic diagram of a comparator in accordance with one embodiment of the present invention.

[0020] Similar elements in Figures are labeled similarly.

DETAILED DESCRIPTION

[0021] Fast access to memory is important for the satisfactory operation of computer systems. As described above, a memory management unit (MMU) conventionally provides translation of virtual addresses into physical addresses utilizing a table defining the relationship between the virtual address generated by a software process and the corresponding physical address. To increase the speed of memory accesses, the MMU provides a cache of the most recently mapped virtual addresses, called a table look-aside buffer (TLB). It is desirable to make the operations utilizing the translation look-aside buffer as fast as possible.

[0022] In one embodiment of the present invention, a compare system for accessing a translation look-aside buffer is described. In this embodiment, the relationship between two operands and their sum is exploited to decrease the time required for determining a match between an applied virtual address and a stored virtual page number. Specifically, the comparison of the sum of two operands to a matching virtual page number entry in a translation look-aside buffer is represented by Equation 1:

$$A + B = K \qquad \text{Equation 1.}$$

where A includes the most significant bits of a first operand, B includes the most significant bits of a second operand, and K is a pre-stored virtual page number from a translation look-aside buffer that matches the sum of operand portion A and operand portion B. Equation 1 may be rewritten to state:

$$A + B + (-K) = 0 \qquad \text{Equation 2.}$$

[0023] In 2's compliment mathematics, the negative of a binary number is equal to the logical NOT of the number (the inversion of each bit in the number) plus 1. Equation 3 represents the 2's compliment of pre-stored virtual page number K:

$$-K = \text{NOT}(K) + 1 \qquad \text{Equation 3.}$$

where -K is the 2's compliment (i.e. the negative) of pre-stored virtual page number K, NOT(K) is the bit inversion (i.e. the logical NOT) of pre-stored virtual page number K, and 1 is a binary 1 padded with enough leading zeroes to match the size of pre-stored virtual page number K.

[0024] Combining Equation 2 and Equation 3 produces:

$$A + B + \text{NOT}(K) + 1 = 0 \qquad \text{Equation 4.}$$

and moving the 1 to the right hand side of Equation 4 produces:

$$A + B + NOT(K) = -1$$ Equation 5.

[0025] The sum of three binary numbers may be represented by a bitwise concatenated sum SUM defined as the concatenation of the bitwise sum SUM(i) bits and a bitwise concatenated carry CARRY defined as the concatenation of the bitwise carry CARRY(i) bits. Sum SUM(i) and carry CARRY(i) are defined as:

$$SUM(i) = A(i)\ XOR\ B(i)\ XOR\ NOT(K(i))$$ Equation 6.

$$CARRY(i) = [A(i-1)\ AND\ B(i-1)]\ OR\ [A(i-1)\ AND\ NOT(K(i-1))]\ OR\ [B(i-1)\ AND\ NOT(K(i-1))]$$ Equation 7.

where i is the i^{th} bit and CARRY(0) is defined as 1 (assuming A + B = K). Therefore, Equation 5 may be re-written as:

$$SUM + CARRY = -1$$ Equation 8.

which may be re-written as:

$$SUM = -1 + -CARRY$$ Equation 9.

[0026] The 2's compliment of carry CARRY is defined as:

$$-CARRY = NOT(CARRY) + 1$$ Equation 10.

which may be combined with Equation 9 to produce:

$$SUM = -1 + NOT(CARRY) + 1$$ Equation 11.

which may be re-written as:

$$SUM = NOT(CARRY)$$ Equation 12.

Thus, Equation 12 shows that if the sum of two operand portions A and B matches a pre-stored virtual page number K, then the concatenated bitwise sums of operand portion A, operand portion B, and the logical NOT of pre-stored virtual page number K will match the logical NOT of the concatenated bit carries of operand portion A, operand portion B, and the logical NOT of pre-stored virtual page number K.

[0027] Figure 2A is a block diagram of a compare system 200 for an MMU used to implement the principle of Equations 1 and 12 in accordance with one embodiment of the present invention. Compare system 200 includes a bit adder 210, comparator 220, and bitwise inverters 231 and 232. Bitwise inverter 231 is used to invert each bit of the pre-stored virtual page number K. Bit adder 210 performs a bitwise addition of operand OPA, operand OPB and the inverted pre-stored virtual page number K to produce a bitwise concatenated sum SUM and a bitwise concatenated carry CARRY. The bitwise addition performed by bit adder 210 is much faster than serial addition of the same operands. Bitwise inverter 231 is used to invert each bit of the bitwise concatenated carry CARRY. Comparator 220 compares each bit of bitwise concatenated sum SUM and the corresponding bit of bitwise concatenated carry NOT(CARRY) to produce a match bit MATCH.

[0028] In one embodiment, comparator 220 is an AND tree, wherein each bit of bitwise concatenated sum SUM and bitwise concatenated carry NOT(CARRY) applied to each corresponding first level AND gate and each first level AND gate output provided to a second level AND gate to produce the match bit MATCH. Figure 2B is a block diagram of an AND tree comparator for a 32 bit address in accordance with one embodiment of the present invention. Each bit of bitwise concatenated sum SM is applied to an AND gate along with each corresponding bit of bitwise concatenated carry

NOT(CARRY). When A+B=K, CARRY(0)=0, so NOT(CARRY(0))=1. Because N=32 (32 bit address) and the bit count starts at "0" and ends at "31", the most significant bit applied to the last first level AND gate (e.g. AND gate 225) is the 31st bit (i.e. N-1). If each bit of bitwise concatenated sum SUM matches each bit of bitwise concatenated carry NOT(CARRY), then the output terminals of AND gates 221-225 (as well as the output terminals of AND gates not shown) provide logic "1" values to AND gate 226. Thus, match bit MATCH is equal to a logic "1" value if the sum of operands OPA and OPB is equivalent to pre-stored virtual page number K.

[0029] Bit adder 210 (Figure 2A) includes a plurality of one-bit adders. Figure 2C is a block diagram for a one-bit adder 211 in accordance with one embodiment of the present invention. One-bit adder 211 receives operand bit OPA(i), the i^{th} bit of operand OPA, operand bit OPB(i), the i^{th} bit of operand OPB, and bit NOT(K(i)), the i^{th} bit of inverted pre-stored virtual page number K. Bit adder produces bitwise sum bit S(i), the i^{th} bit of the bitwise concatenated sum SUM, in accordance with Equation 6, and bitwise concatenated carry bit CARRY(i+1), the $(i+1)^{th}$ bit of bitwise concatenated carry CARRY, in accordance with Equation 7.

[0030] One-bit adder 211 produces a bitwise sum bit SUM(i) and a bitwise carry bit CARRY(i) from the applied data independently of the function of other one-bit adders comprising bitwise adder 210. While the bitwise concatenated sum SUM and bitwise concatenated carry CARRY need to be further manipulated to produce a true sum (i.e. a word sum of the applied data) and a true carry (i.e. a word carry bit after generating the word sum), the bitwise concatenated sum and bitwise concatenated carry produced by bitwise adder 210 are sufficient to utilize the properties of Equations 6, 7, and 12 to determine whether a match exists between the sum of the two operands and the pre-stored virtual page number. Thus, bitwise adder 210, unlike a word adder, may perform the add of each bit of the applied data in parallel. As a result, the time required to generate an N bit concatenated sum SUM is many times faster than the time required to generate an N bit word sum.

[0031] A compare system similar to compare system 200 (Figure 2A) is provided for each virtual page number (i.e. pre-stored virtual page number K) in the translation look-aside buffer of an MMU. Therefore, if a translation look-aside buffer has four rows, and thus stores four virtual page numbers, the MMU will have four compare systems similar to compare system 200, one for each row. If a particular compare system 200 has an output bit (i.e. match bit MATCH) equal to a logic "1" value, then the associated physical page number for that pre-stored virtual page number K is provided as the output physical page number of the translation look-aside buffer.

[0032] In another embodiment of the present invention, the page size of the MMU is used to truncate the number of bits applied to a bit adder similar to bit adder 210 of Figure 2A. For example, in a system with 32 bit addresses, a 1K page size results in the most significant 22 bits of the address corresponding to the virtual page number and the least significant 10 bits of the address corresponding to the page offset. As a result, only the most significant 22 bits need be used for a compare system similar to compare system 200 of Figure 2A. However, a compare system according to this embodiment requires a carry bit from the word sum of the least significant 10 bits of the address as described below.

[0033] Figure 3 is a block diagram of a compare system 300 in accordance with another embodiment of the present invention. In this example of compare system 300, a 64K page size and 32 bit address is described. Thus, the virtual page number consists of the 16 most significant bits of the sum of operands OPA and OPB.

For clarity, the 16 most significant bits of operand OPA are called an operand portion OPA_H and the 16 most significant bits of operand OPB are called an operand portion OPB_H. Similarly, the 16 least significant bits of operand OPA and OPB are called operand portions OPA_L and OPB_L, respectively. Compare system 300 includes bit adder 310 that receives operand portions OPA_H and OPB_H as well as a logical NOT of a pre-stored virtual page number K (through bitwise inverter 331). Bit adder 310 provides a 16 bit bitwise concatenated sum SUM and a 16 bit bitwise concatenated carry CARRY according to Equations 6 and 7, respectively.

[0034] As described above, comparator 320 receives bitwise concatenated sum SUM and the logical NOT of bitwise concatenated carry CARRY (through bitwise inverter 331). While comparing the least significant bit of bitwise concatenated sum SUM (e.g., the 0^{th} bit) to the opposite of the least significant bit of bitwise concatenated carry CARRY (the 0^{th} bit), Equation 7 shows that the 0^{th} bit of bitwise concatenated carry CARRY can not be calculated from the operand portions applied to bit adder 310. Therefore, comparator 320 must also receive a wordwise carry bit calculated from operand portions OPA_L and OPB_L to define this 0^{th} bit of bitwise concatenated carry CARRY. Operand portions OPA_L and OPB_L are applied to word adder 340. Word adder 340 generates a 16 bit wordwise sum WSUM of those operand portions as well as a wordwise carry bit C_OUT. Wordwise sum WSUM is a true sum of operand portions OPA_L and OPB_L. Wordwise carry bit C_OUT is inverted by inverter 333 and applied to comparator 320 for comparison to the 0^{th} bit of bitwise concatenated sum SUM. Comparator 320 generates a one bit match signal MATCH which has a logic "1" value if the virtual page number generated by the true sum of operands OPA and OPB matches the pre-stored virtual page number K, and has a logic "0" value otherwise. While compare system 300 has been described with respect to a 64K page size system and 32 bit addresses, one skilled in the art can easily adapt compare system 300 to other page sizes (e.g. a 1K, 4K, and 16K page size) and other addresses sizes (e.g. 16 bits). For example, a 16K page size and 32 bit address requires a 14 bit page offset with 18 bits of the 32 bit address left over for the virtual page number.

[0035] For some applications, for example, the page size is a selectable variable that may be defined by the software. For these types of applications, the size of the virtual page numbers vary depending on the definition of the page size variable. Thus, some embodiments of the present invention include a variable size comparison system to accommodate the variable page size. While the present embodiment is described with respect to memory management units, the principles of this embodiment may be adapted to any application utilizing comparisons of variable sized operands.

[0036] Figure 4A is a block diagram of a compare system 400 in accordance with an embodiment of the present invention. Compare system 400 is configured for 32 bit addresses and a variable page size of 1K, 4K, 16K or 64K. One skilled in the art may easily adapt this embodiment to other ranges of address size and page size. Bit adder 410 receives an operand portion OPA_H, an operand portion OPB_H, and the logical NOT of a pre-stored virtual page number K through inverter 431. The length of K is implemented in the hardware to be the maximum possible virtual page number. In this case, the maximum possible virtual page number is for a 1K page size (2^{10}), which leaves 32 bits (address size) less 10 bits (1K page size), or 22 bits. Operand portions OPA_H and OPB_H are the most significant 22 bits of operands OPA and OPB, respectively, for similar reasons. Bit adder 410 provides a 22 bit bitwise concatenated sum S and a 22 bit bitwise concatenated carry CARRY. The logical NOT of bitwise concatenated carry CARRY, bitwise concatenated carry CB, is formed using bitwise inverter 432.

[0037] Word adder 440 receives an augmented operand portion AOPA_L and an augmented operand portion AOPB_L. In this example, augmented operand portion AOPA_L is a 19 bit number which is formed from the 16 least significant bits of operand OPA augmented with three logic "1" indicator bit values in the following manner:

OPA[15:14],1,OPA[13:12],1,OPA[11:10],1,OPA[9:0]

For example, the 18th bit of augmented operand AOPA_L is equal to the 15th bit of operand OPA, the 16th bit of augmented operand AOPA_L is equal to a logic "1" indicator bit value, and the 14th bit of augmented operand AOPA_L is equal to the 12th bit of operand OPA. Similarly, augmented operand portion AOPB_L is a 19 bit number which is formed from the 16 least significant bits of operand OPB augmented with three logic "0" indicator bit values in the following manner:

OPB[15:14],0,OPB[13:12],0,OPB[11:10],0,OPB[9:0]

For example, the 16th bit of augmented operand AOPB_L is equal to a logic "0" indicator bit value, the 11th bit of augmented operand AOPB_L is equal to the 10th bit of operand OPB, and the 0th bit of augmented operand AOPB_L is equal to the 0th indicator bit bit of operand OPB.

[0038] Word adder 440 adds together augmented operands AOPA_L and AOPB_L, producing a 19 bit augmented wordwise sum SUM and a one bit wordwise carry C_OUT. Wordwise carry C_OUT is inverted by inverter 433. Comparator 420 receives a select signal SEL that indicates the length of the page size. The select signal SEL is used to choose among the indicator bits from wordwise sum SUM and to compare that chosen indicator bit to the proper bit of bitwise concatenated sum S. Comparator 420 receives bitwise concatenated carry CB from inverter 432, bitwise concatenated sum S from bit adder 410, augmented wordwise sum SUM from word adder 440, the carry CB from inverter 433, and the select signal SEL, producing a match bit MATCH. If match MATCH has a logic "1" value, then the sum of operands OPA and OPB matches pre-stored virtual page number K. If match bit MATCH has a logic "0" value, then the virtual page number formed from the sum of the virtual page number from the operands OPA and OPB does not match pre-stored virtual page number K. This operation is described in more detail below.

[0039] Figure 4B is an example of an augmentation and then addition of two numbers (operand portions) OP1 and OP2 producing a wordwise sum ASUM and a wordwise carry bit ACARRY. Augmented numbers AUG1 and AUG2 are generated from operand portions OP1 and OP2, respectively, similar to augmented operand portions AOPA_L and AOPB_L. The bit positions of the numbers used in the calculation are labeled on line 450. The bit values of operand portion OP1 are shown on line 451, each bit value corresponding to a bit position on line 450. These bit values are augmented with logic "1" values at bit positions AUG1[16], AUG1[13], and AUG1[10] as shown on line 452. The bit values of operand portion OP2 are shown on line 453, each bit value corresponding to a bit position on line 450. These values are augmented with logic "0" values at bit positions AUG2[16], AUG2[13], and AUG2[10] as shown on line 454. The bit positions (line 450), augmented operand portion AUG1 (line 452), and augmented operand portion AUG2 (line 454) are reproduced on lines 455, 456, and 457, respectively. The wordwise sum (i.e. the true sum) of augmented operand portions AUG1 and AUG2 is shown as wordwise sum ASUM on line 458 and the wordwise carry bit ACARRY shown on line 459. The wordwise sum of augmented operand portions AUG1[9:0] and AUG2[9:0] produces no carry bit. As a result, the bit in bit position 10 has a logic "1" value. Similarly, the word sum of augmented operand portions AUG1[12:0] and AUG2[12:0] produces no carry bit. Therefore, the bit in bit position 13 has a logic "1" value. In contrast, the word sum of augmented operand portions AUG1[15:0] and AUG2[15:0] does produce a carry bit. As a result, the bit in bit position 16 has a logic "0" value. The value stored in bit positions 10, 13, and 16 (i.e. the indicator bits) are used to indicate the carry from the less significant bits of operands OP1 and OP2 in comparator 420. Comparator 420 is described in more detail below.

[0040] Figure 5 is a schematic diagram of a comparator 500 in accordance with one embodiment of the present invention. Each bit in bitwise concatenated sum S is compared to a corresponding carry bit, either in bitwise concatenated carry CB, in wordwise carry C, or an indicator bit in wordwise sum SUM as described in Equations 6, 7, and 12 above. This is accomplished with the use of a two input XNOR gate (an inverted output exclusive-OR), which produces a logic "1" value when the two applied values match. Thus, comparator 500 utilizes XNOR gates 501-510 (and additional XNOR gates, not shown) to each compare two bits. For example, XNOR gate 501 compares a bit of bitwise concatenated sum S (S[21]) to a bit of bitwise concatenated carry CB (CB[20]) and produces a logic "1" value if bit S[21] matches bit CB[20]. The most significant 15 bits of bitwise concatenated sum S (i.e. bits S[21:7]) and the corresponding bits of bitwise concatenated carry CB (i.e. bits CB[20:6]) are similarly compared using an XNOR gate, because these bits remain a part of the comparison operation for all page sizes in the present embodiment. The output terminals of XNOR gates 501-504 (as well as the XNOR gates not shown) are coupled to input terminals of AND gate 530. The output terminal of AND gate 530 is coupled to a first input terminal of AND gate 537. AND gate 537 provides a logic "1" value match bit MATCH if the sum of the operands OPA and OPB (Figure 3A) match the pre-stored virtual page number K, and provides a logic "0" value match bit MATCH otherwise.

[0041] When the page size is chosen to be 64K, then the bits of interest to the virtual page number comparison are the most significant 16 bits of the bitwise concatenated sum bit S, bitwise concatenated carry bit CB, and wordwise carry bit C. If the chosen page size is 64K, XNOR gate 504 compares the wordwise carry bit C to bitwise concatenated sum bit S[6]. However, if the chosen page size is smaller than 64K (e.g. 16K), then XNOR gate 504 compares the bitwise concatenated carry bit CB[5] to bitwise concatenated sum bit S[6]. As a result, bitwise concatenated carry bit CB[5] and wordwise carry bit C are applied to the input data terminals of a multiplexer 541 using a selector bit SEL[3] applied to a control terminal of multiplexer 541. As a result, one of bitwise concatenated carry bit CB[5] and wordwise carry C are applied to an input terminal of XNOR gate 504 along with bitwise concatenated sum bit S[6]. Selector bit SEL[3] has a logic "1" value when the chosen page size is 64K, passing wordwise carry bit C to XNOR gate 504. This logic "1" value of selector bit SEL[3] is also applied to OR gate 541, causing a logic "1" value to be applied at a second input terminal of AND gate 537. As a result, when selector bit SEL[3] has a logic "1" value, if bitwise concatenated sum bits S[21:7] match bitwise concatenated carry bits CB[20:6] and bitwise concatenated sum bit S[6] matches wordwise carry bit C, then AND gate 537 provides a logic "1" match value MATCH.

[0042] If the chosen page size is 16K or smaller, then XNOR gate 505 compares bitwise concatenated sum bit S[5] to bitwise concatenated carry bit CB[4]. Additionally, XNOR gate 506 compares either bitwise concatenated carry bit CB[3] or indicator bit SUM[16] to bitwise concatenated sum bit S[6]. The output terminals of XNOR gates 505 and 506 are coupled to input terminals of AND gate 531, which has an output terminal coupled to a first input terminal of AND gate 534. A logic "1" value of selector bit SEL[2] applied to the control input terminal of multiplexer 542 causes indicator bit SUM[16] to be compared to bitwise concatenated carry bit CB[3].

[0043] If the chosen page size is 16K, then XNOR gate 506 compares indicator bit ASUM[16] to bitwise concatenated sum bit S[6]. However, if the chosen page size is smaller than 16K (e.g. 4K and selector bit SEL[2] has a logic "0" value), then XNOR gate 506 compares bitwise concatenated carry bit CB[3] to bitwise concatenated sum bit S[4]. As described above, bitwise concatenated carry bit CB[3] and augmented sum carry bit SUM[16] are applied to input data terminals of a multiplexer 542 using the selector bit SEL[2] applied to the control terminal of multiplexer 542. Selector bit SEL[2] has a logic "1" value when the chosen page size is 16K, passing indicator bit SUM[16] to XNOR gate 506. This logic "1" value of selector bit SEL[2] is also applied to a first input terminal of AND gate 534, causing the value provided at the output terminal of AND gate 531 to pass through AND gate 534 to a second input terminal of OR gate 541. As a result, if bitwise concatenated sum bits S[21:5] match bitwise concatenated carry bits CB[20:4] and bitwise concatenated sum bit S[4] matches indicator bit SUM[16], then AND gate 537 provides a logic "1" match value bit MATCH. The 1K and 4K page sizes are similarly calculated. An embodiment similar to that described with respect to Figure 5 is shown in Appendix I.

In the various embodiments of this invention, novel structures and methods have been described to speed comparison of virtual page numbers in MMUs as well as to speed variable-sized number comparison. Using a comparator in place of large adder in accordance with an embodiment of the present invention, the delay of a serial bit adder can be avoided by performing a bit by bit comparison utilizing a mathematical relationship between matching words. Additionally, utilizing a smaller word adder and an augmented portion of applied words in accordance with another embodiment of the present invention, comparison speed may be improved while accommodating variably sized input words. The various embodiments of the structures and methods of this invention that are described above are illustrative only of the principles of this invention and are not intended to limit the scope of the invention to the particular embodiments described. For example, in view of this disclosure, those skilled in the art can define other page sizes, address sizes, component implementations, contexts, and so forth, and use these alternative features to create a method or system according to the principles of this invention. Thus, the invention is limited only by the following claims.

APPENDIX I

```
////////////////////////////////////////////////////////////////////////////////
//                                                                            //
//                       #######  #####   #####                               //
//                          #    #     # #     #                              //
//                          #    #           #                                //
//                          #    #      #####                                 //
//                          #    #     #                                      //
//                          #    #     # #                                    //
//                          #     #####  #######                              //
//                                                                            //
//     File Name :- mmu_mtlb.v                                                //
//   Description :- Verilog coding for micro tlb for IMTLB and DMTLB          //
//       Creator :- Park, Heonchul                                            //
//          Date :- March 2001                                                //
//       Version :- 0.1                                                       //
//       History :- Initial Version                                           //
//           Copyright(c) Infineon AG 2000 all rights reserved                //
////////////////////////////////////////////////////////////////////////////////


module mmu_mtlb(clk, testmode, base_addr, offset_addr, tte_psz,tte_asi,
        tte_vpn, tte_ppn, tte_xe, tte_we, tte_spr,
      tte_re, tte_c, tte_g, tte_v, tte_write, we_sel, addr_in, asi_in,
        mmu_con_psza, mmu_con_pszb, psw_su, mtlb_sel, utlb_sel,
      flush, mtlb_addr, mtlb_we, mtlb_re, mtlb_xe, mtlb_hit, mtlb_c,
        mtlb_psz, mtlb_spr,
      vpn0_match, vpn1_match, vpn2_match, vpn3_match);

input clk;
input testmode;
input [1:0] tte_psz;
input [4:0] tte_asi;
input [21:0] tte_vpn;
input [21:0] tte_ppn;
input tte_xe;
input [1:0]  tte_spr;
input tte_we;
input tte_re;
input [1:0] tte_c; //{suser, tte_c}
input tte_g;
input tte_v;
input tte_write;
input [1:0] we_sel;
input [21:0] addr_in;
input [4:0] asi_in;
input [1:0] mmu_con_psza;
input [1:0] mmu_con_pszb;
input psw_su;
input flush;
input [31:0] base_addr;
input [31:0] offset_addr;
input mtlb_sel;
input utlb_sel;
output [21:0] mtlb_addr;
output mtlb_hit;
output mtlb_xe, mtlb_re, mtlb_we;
output mtlb_c;
output [1:0] mtlb_psz;
output  vpn0_match, vpn1_match, vpn2_match, vpn3_match;
output [1:0] mtlb_spr;

reg [1:0] psz0, psz1, psz2, psz3;
reg [4:0] asi0, asi1, asi2, asi3;
reg [21:0] vpn0, vpn1, vpn2, vpn3;
reg [21:0] ppn0, ppn1, ppn2, ppn3;
reg [1:0] spr0, spr1, spr2, spr3;
reg xe0, xe1, xe2, xe3;
reg we0, we1, we2, we3;
reg re0, re1, re2, re3;
reg [1:0] c0, c1, c2, c3;
reg g0, g1, g2, g3;
reg v0, v1, v2, v3;

wire [21:0] mtlb_addr;
wire [1:0]  mtlb_psz;
wire     mtlb_xe, mtlb_re, mtlb_we;
wire     mtlb_c;


// write into latches on low phase clock if tte_write is high
wire latch0_sel = (~clk & tte_write & ~we_sel[1] & ~we_sel[0]) ;
wire latch1_sel = (~clk & tte_write & ~we_sel[1] &  we_sel[0]) ;
wire latch2_sel = (~clk & tte_write &  we_sel[1] & ~we_sel[0]) ;
wire latch3_sel = (~clk & tte_write &  we_sel[1] &  we_sel[0]) ;

always @(latch0_sel or  tte_psz or tte_asi or tte_vpn or tte_ppn or
      tte_xe or tte_we or tte_re or tte_c or tte_g or tte_spr)
if (latch0_sel)
begin
      psz0 <= tte_psz;
      asi0 <= tte_asi;
      vpn0 <= tte_vpn;
      ppn0 <= tte_ppn;
        spr0 <= tte_spr;
      xe0 <= tte_xe;
      we0 <= tte_we;
      re0 <= tte_re;
      c0 <= tte_c;
      g0 <= tte_g;
end
always @(latch1_sel or  tte_psz or tte_asi or tte_vpn or tte_ppn or
      tte_xe or tte_we or tte_re or tte_c or tte_g or tte_spr)
if (latch1_sel)
begin
      psz1 <= tte_psz;
      asi1 <= tte_asi;
      vpn1 <= tte_vpn;
      ppn1 <= tte_ppn;
        spr1 <= tte_spr;
      xe1 <= tte_xe;
      we1 <= tte_we;
      re1 <= tte_re;
      c1 <= tte_c;
      g1 <= tte_g;
```

```
end
always @(latch2_sel or  tte_psz or tte_asi or tte_vpn or tte_ppn or
      tte_xe or tte_we or tte_re or tte_c or tte_g or tte_spr)
if (latch2_sel)
begin
      psz2 <= tte_psz;
      asi2 <= tte_asi;
      vpn2 <= tte_vpn;
      ppn2 <= tte_ppn;
        spr2 <= tte_spr;
      xe2 <= tte_xe;
      we2 <= tte_we;
      re2 <= tte_re;
      c2 <= tte_c;
      g2 <= tte_g;
end
always @(latch3_sel or  tte_psz or tte_asi or tte_vpn or tte_ppn or
      tte_xe or tte_we or tte_re or tte_c or tte_g or tte_spr)
if (latch3_sel)
begin
      psz3 <= tte_psz;
      asi3 <= tte_asi;
      vpn3 <= tte_vpn;
      ppn3 <= tte_ppn;
        spr3 <= tte_spr;
      xe3 <= tte_xe;
      we3 <= tte_we;
      re3 <= tte_re;
      c3 <= tte_c;
      g3 <= tte_g;
end



always @(clk or flush or we_sel or tte_write or tte_v)
if (~clk)
   if (flush)   begin
         v0 <= 1'b0;
         end
   else if (tte_write & ~we_sel[1] & ~we_sel[0]) begin
         v0 <= tte_v;
         end

always @(clk or flush or we_sel or tte_write or tte_v)
if (~clk)
   if (flush)     begin
         v1 <= 1'b0;
         end
   else if (tte_write & ~we_sel[1] & we_sel[0]) begin
         v1 <= tte_v;
         end

always @(clk or flush or we_sel or tte_write or tte_v)
if (~clk)
   if (flush)     begin
         v2 <= 1'b0;
         end
   else if (tte_write & we_sel[1] & ~we_sel[0]) begin
            v2 <= tte_v;
            end

always @(clk or flush or we_sel or tte_write or tte_v)
if (~clk)
   if (flush)     begin
            v3 <= 1'b0;
            end
   else if (tte_write & we_sel[1] & we_sel[0]) begin
            v3 <= tte_v;
            end

// compare vpn0 with operanda + operandb
wire [21:0] vpn0_inv = ~vpn0;
wire [21:0] vpn1_inv = ~vpn1;
wire [21:0] vpn2_inv = ~vpn2;
wire [21:0] vpn3_inv = ~vpn3;

wire [21:0] vpn0_carry = ~((base_addr[31:10] & offset_addr[31:10]) |
                   (base_addr[31:10] & vpn0_inv) |
                   (vpn0_inv & offset_addr[31:10]));

wire [21:0] vpn1_carry = ~((base_addr[31:10] & offset_addr[31:10]) |
                   (base_addr[31:10] & vpn1_inv) |
                   (vpn1_inv & offset_addr[31:10]));

wire [21:0] vpn2_carry = ~((base_addr[31:10] & offset_addr[31:10]) |
                   (base_addr[31:10] & vpn2_inv) |
                   (vpn2_inv & offset_addr[31:10]));

wire [21:0] vpn3_carry = ~((base_addr[31:10] & offset_addr[31:10]) |
                   (base_addr[31:10] & vpn3_inv) |
                   (vpn3_inv & offset_addr[31:10]));
wire [21:0] vpn0_sum = (base_addr[31:10] ^ offset_addr[31:10] ^ vpn0_inv);
wire [21:0] vpn1_sum = (base_addr[31:10] ^ offset_addr[31:10] ^ vpn1_inv);
wire [21:0] vpn2_sum = (base_addr[31:10] ^ offset_addr[31:10] ^ vpn2_inv);
wire [21:0] vpn3_sum = (base_addr[31:10] ^ offset_addr[31:10] ^ vpn3_inv);

wire [19:1] lower_adder = {1'b1, base_addr[15:14], 1'b1, base_addr[13:12],
                     1'b1, base_addr[11:10], 1'b1, base_addr[9:1]} +
                   {1'b0, offset_addr[15:14], 1'b0, offset_addr[13:12],
                 1'b0, offset_addr[11:10], 1'b0, offset_addr[9:1]};

wire match0_21_7 = &(vpn0_carry[20:6] ^~ vpn0_sum[21:7]);
wire match0_6    = (psz0 == 2'b11)? (lower_adder[19] ^~ vpn0_sum[6]):
                               (vpn0_sum[6] ^~ vpn0_carry[5]);
wire match0_21_6 = match0_21_7 & match0_6;
wire match0_16k = (psz0 == 2'b10)? lower_adder[16]: vpn0_carry[3];
wire match0_5_4 = &(vpn0_sum[5:4] ^~ {vpn0_carry[4],match0_16k});
wire match0_4k = (psz0 == 2'b01)? lower_adder[13]: vpn0_carry[1];
wire match0_3_2 = &(vpn0_sum[3:2] ^~ {vpn0_carry[2],match0_4k});
wire match0_1k = (psz0 == 2'b00)? lower_adder[10]: 1'b1;
wire match0_1_0 = &(vpn0_sum[1:0] ^~ {vpn0_carry[0],match0_1k});
wire vpn0_hit = match0_21_6 &
      ((psz0[1] & psz0[0]) | (match0_5_4 & psz0[1] & ~psz0[0]) |
      ((~psz0[1] & psz0[0]) & match0_5_4 & match0_3_2) |
      (match0_5_4 & match0_3_2 & match0_1_0));


wire match1_21_7 = &(vpn1_carry[20:6] ^~ vpn1_sum[21:7]);
wire match1_6    = (psz1 == 2'b11)? (lower_adder[19] ^~ vpn1_sum[6]):
                                    (vpn1_sum[6] ^~ vpn1_carry[5]);
wire match1_21_6 = match1_21_7 & match1_6;
wire match1_16k = (psz1 == 2'b10)? lower_adder[16]: vpn1_carry[3];
wire match1_5_4 = &(vpn1_sum[5:4] ^~ {vpn1_carry[4],match1_16k});
wire match1_4k = (psz1 == 2'b01)? lower_adder[13]: vpn1_carry[1];
wire match1_3_2 = &(vpn1_sum[3:2] ^~ {vpn1_carry[2],match1_4k});
wire match1_1k = (psz1 == 2'b00)? lower_adder[10]: 1'b1;
wire match1_1_0 = &(vpn1_sum[1:0] ^~ {vpn1_carry[0],match1_1k});
wire vpn1_hit = match1_21_6 &
      ((psz1[1] & psz1[0]) | (match1_5_4 & psz1[1] & ~psz1[0]) |
      ((~psz1[1] & psz1[0]) & match1_5_4 & match1_3_2) |
      (match1_5_4 & match1_3_2 & match1_1_0));

wire match2_21_7 = &(vpn2_carry[20:6] ^~ vpn2_sum[21:7]);
wire match2_6    = (psz2 == 2'b11)? (lower_adder[19] ^~ vpn2_sum[6]):
                                    (vpn2_sum[6] ^~ vpn2_carry[5]);
wire match2_21_6 = match2_21_7 & match2_6;
wire match2_16k = (psz2 == 2'b10)? lower_adder[16]: vpn2_carry[3];
wire match2_5_4 = &(vpn2_sum[5:4] ^~ {vpn2_carry[4],match2_16k});
wire match2_4k = (psz2 == 2'b01)? lower_adder[13]: vpn2_carry[1];
wire match2_3_2 = &(vpn2_sum[3:2] ^~ {vpn2_carry[2],match2_4k});
wire match2_1k = (psz2 == 2'b00)? lower_adder[10]: 1'b1;
wire match2_1_0 = &(vpn2_sum[1:0] ^~ {vpn2_carry[0],match2_1k});
wire vpn2_hit = match2_21_6 &
      ((psz2[1] & psz2[0]) | (match2_5_4 & psz2[1] & ~psz2[0]) |
      ((~psz2[1] & psz2[0]) & match2_5_4 & match2_3_2) |
      (match2_5_4 & match2_3_2 & match2_1_0));


wire match3_21_7 = &(vpn3_carry[20:6] ^~ vpn3_sum[21:7]);
wire match3_6    = (psz3 == 2'b11)? (lower_adder[19] ^~ vpn3_sum[6]):
                                    (vpn3_sum[6] ^~ vpn3_carry[5]);
wire match3_21_6 = match3_21_7 & match3_6;
wire match3_16k = (psz3 == 2'b10)? lower_adder[16]: vpn3_carry[3];
wire match3_5_4 = &(vpn3_sum[5:4] ^~ {vpn3_carry[4],match3_16k});
wire match3_4k = (psz3 == 2'b01)? lower_adder[13]: vpn3_carry[1];
wire match3_3_2 = &(vpn3_sum[3:2] ^~ {vpn3_carry[2],match3_4k});
wire match3_1k = (psz3 == 2'b00)? lower_adder[10]: 1'b1;
wire match3_1_0 = &(vpn3_sum[1:0] ^~ {vpn3_carry[0],match3_1k});
wire vpn3_hit = match3_21_6 &
      ((psz3[1] & psz3[0]) | (match3_5_4 & psz3[1] & ~psz3[0]) |
      ((~psz3[1] & psz3[0]) & match3_5_4 & match3_3_2) |
      (match3_5_4 & match3_3_2 & match3_1_0));



// generate matching output

wire valid0 = v0 & ((asi0 == asi_in) | g0) & ((psz0 == mmu_con_psza) | (psz0 ==
mmu_con_pszb));
wire vpn0_match = valid0 & vpn0_hit;
wire valid1 = v1 & ((asi1 == asi_in) | g1) & ((psz1 == mmu_con_psza) | (psz1 ==
mmu_con_pszb));
wire vpn1_match = valid1 & vpn1_hit;

wire valid2 = v2 & ((asi2 == asi_in) | g2) & ((psz2 == mmu_con_psza) | (psz2 ==
mmu_con_pszb));
wire vpn2_match = valid2 & vpn2_hit;

wire valid3 = v3 & ((asi3 == asi_in) | g3) & ((psz3 == mmu_con_psza) | (psz3 ==
mmu_con_pszb));
wire vpn3_match = valid3 & vpn3_hit;

// output generation



wire mtlb_hit = vpn0_match | vpn1_match | vpn2_match | vpn3_match;

MMU_MUXH_4 #(1) mtlb_xe_mux (.d0(xe0), .d1(xe1), .d2(xe2), .d3(xe3),
                        .s0(vpn0_match), .s1(vpn1_match), .s2(vpn2_match),
.s3(vpn3_match),
                       .out(mtlb_xe));
MMU_MUXH_4 #(1) mtlb_re_mux (.d0(re0), .d1(re1), .d2(re2), .d3(re3),
                        .s0(vpn0_match), .s1(vpn1_match), .s2(vpn2_match),
.s3(vpn3_match),
                       .out(mtlb_re));
MMU_MUXH_4 #(1) mtlb_we_mux (.d0(we0), .d1(we1), .d2(we2), .d3(we3),
                        .s0(vpn0_match), .s1(vpn1_match), .s2(vpn2_match),
.s3(vpn3_match),
                       .out(mtlb_we));
wire c0_in = c0[0] & (c0[1] == psw_su);
wire c1_in = c1[0] & (c1[1] == psw_su);
wire c2_in = c2[0] & (c2[1] == psw_su);
wire c3_in = c3[0] & (c3[1] == psw_su);
wire mtlb_psz0_sel = vpn0_hit & mtlb_sel;
wire mtlb_psz1_sel = vpn1_hit & mtlb_sel;
wire mtlb_psz2_sel = vpn2_hit & mtlb_sel;
wire mtlb_psz3_sel = vpn3_hit & mtlb_sel;

MMU_MUXH_4 #(1) mtlb_c_mux (.d0(c0_in), .d1(c1_in), .d2(c2_in), .d3(c3_in),
                       .s0(vpn0_hit), .s1(vpn1_hit), .s2(vpn2_hit), .s3(vpn3_hit),
                       .out(mtlb_c));

MMU_MUXH_5 #(2) mtlb_psz_mux (.d0(psz0), .d1(psz1), .d2(psz2), .d3(psz3),
              .d4(tte_psz),  .s0(mtlb_psz0_sel), .s1(mtlb_psz1_sel),
                     .s2(mtlb_psz2_sel), .s3(mtlb_psz3_sel), .s4(utlb_sel),
                                .out(mtlb_psz));

MMU_MUXH_4 #(2) mtlb_spr_mux (.d0(spr0), .d1(spr1), .d2(spr2), .d3(spr3),
                       .s0(vpn0_hit), .s1(vpn1_hit), .s2(vpn2_hit), .s3(vpn3_hit),
                               .out(mtlb_spr));

MMU_MUXH_4 #(22) mtlb_mux (.d0(ppn0), .d1(ppn1), .d2(ppn2), .d3(ppn3),
                       .s0(vpn0_hit), .s1(vpn1_hit), .s2(vpn2_hit), .s3(vpn3_hit),
                       .out(mtlb_addr));
```

```
endmodule
```